OCL_{FO}: First Order Expressive OCL Constraints for Efficient Integrity Checking

Enrico Franconi · Alessandro Mosca · Xavier Oriol · Guillem Rull · Ernest Teniente

Abstract OCL is the standard language for defining constraints in UML class diagrams. Unfortunately, as we show in this paper, full OCL is so expressive that it is not possible to check general OCL constraints efficiently. In particular, we show that checking general OCL constraints is not only not polynomial, but not even semidecidable. To overcome this situation, we identify OCL_{FO}, a fragment of OCL which is expressively equivalent to relational algebra (RA). By equivalent we mean that any OCL_{FO} constraint can be checked through a RA query (which guarantees that OCL_{FO} checking is efficient, i.e., polynomial), and any RA query encoding some constraint can be written as an OCL_{FO} constraint (which guarantees expressiveness of OCL_{FO}). In this paper we define the syntax of OCL_{FO}, we concisely determine its semantics through set theory, and we proof its equivalence to RA. Additionally, we identify the core of this language, i.e. a minimal subset of OCL_{FO} equivalent to RA.

Keywords OCL, relational algebra, translation, integrity checking

1 Introduction

Since the definition of the Entity-Relationship (ER) language by Peter Chen in his seminal paper of 1976 [1], several new graphical modeling languages have been proposed so far by different researchers and institutions. Some prominent examples might be the Object Role Modeling language (ORM)[2], and the Unified Modeling language (UML) [3].

Using these languages, a software engineer can specify the conceptual schema of an information system. That is, the relevant concepts of the domain of interest, and how these concepts are related. Intuitively, the structural part of a conceptual schema (the one considered in this paper) consists of a class diagram, together with a set of textual constraints usually specified in a formal language such as OCL (Object Constraint Language) [4].

For instance, in Figure 1 we show a UML class diagram for some messaging application. The domain of such application consists of users, conversation groups (which can be divided into pairs, and groups) and messages. In this domain, users belong to conversation groups, and messages are sent by users to these groups.

To get a precise conceptual schema, this diagram has to be complemented with a set of constraints, i.e., conditions that the data it specifies should satisfy to be considered valid. Then, modeling languages provide themselves some graphical constructs that allow the definition of some frequent constraints. For instance, in our running example, we have stated the constraint that each message is sent to exactly one conversation group by means of two graphical UML cardinalities.

Moreover, and due to the limited expressiveness of graphical constraints, a textual language has to be considered also to express more sophisticated constraints.
Indeed, constraints such as *Users only send messages to groups they belong to*, and *Messages of a group are sent after the group creation* cannot be graphically expressed in the previous UML diagram.

Currently, OCL (Object Constraint Language) [4] is probably the most popular notation to specify textual constraints and an ISO/IEC standard. Roughly, OCL permits defining constraints by means of building navigations from classes/associations, and applying some OCL operators to such navigations. For instance, the previously mentioned constraints can be specified in OCL as:

```
context Group inv MessagesAreFromGroup:
  self.user->includesAll(self.msg.author)

context Group inv MessagesAreSentAfterCreation:
  self.msg->forAll(m|m.sentTime > self.crTime)
```

Fig. 2 OCL constraints

However a natural question arising is to what extent is OCL expressive for defining constraints. That is, is OCL expressive enough for defining all the constraint we might require? or, on the contrary, is it even excessively expressive? In fact, the expressive power of a language determines the complexity to check or to analyze its expressions [5]. Therefore, we may want to avoid a language which is too expressive because it would penalize the time for its checking, but it may happen also that a restrictive language might be efficiently checked but useless.

This is an important question, which has no answer yet in the OCL literature. Probably, the closest study in this area is the one by Mandel and Cengarle in [6] but it was performed more than 15 years ago, and for an old version of OCL which did not include new constructs and capabilities that have been released since then. This is why we understand that a new and more careful analysis must be done. This is particularly important if we take into account that, in the OCL panel held in the OCL Workshop 2014, the OCL community discussed about whether OCL should include more operations (and which ones), or if OCL should just reorganize the current existing ones [7].

In this context, we show in this paper that full OCL is currently so expressive that it is able to encode non-decidable constraints (and even non-semidecidable ones). That is, we can write OCL constraints for which there is no algorithm able to check its satisfaction in finite time. It is important to highlight that previous results only showed that OCL was undecidable for the problem of satisfiability (that is, whether it exists some data instance satisfying the constraint) [8], but we show that it is even undecidable with the satisfaction problem.

That means that OCL interpreters might not be able to assess whether an OCL constraint is satisfied by some data, even in the case that the data is indeed satisfying the constraint. Clearly, this result has a deep impact in the efficiency of OCL implementations like [9-11]. Indeed, our results not only imply undecidability, but also that decidable cases might be exponential. Thus, despite such implementations give support to most OCL expressions, many of them would only be feasible in toy examples rather than in real datasets. Our article brings light to what set of OCL operations can scale up for being checked with real big datasets.

In particular, in this article we identify a fragment of OCL which is expressive enough to write the most typically used textual constraints, but without loosing good computational properties for checking them. Our key idea is to look for the OCL fragment whose expressions can be checked through relational algebra (RA) queries in the sense that the constraint is violated iff the RA query returns a non-empty answer.

For instance, assume a translation from the UML class diagram in Figure 1 into a relational database such that each class gives rise to a different table with the same name, the same attributes plus an additional one corresponding to the OID (i.e. the object identifier, which is set for each tuple to a fresh value, different from the OID of any other tuple in the whole instance); and such that each association becomes also a table named as the association and with as many attributes as classes participate in the association (i.e. the OIDs of the participating classes).

Then, we have that the OCL constraints in Figure 2 can be checked by means of the following RA queries:

1. \(\pi(\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Writes}) \setminus \pi(\text{Group} \bowtie \text{HasMember})\)

2. \(\sigma_{{\text{sentTime} \leq \text{crTime}}} (\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Message})\)
Intuitively, the first query looks for the users which send messages to groups they do not belong to. In this way, one can check if the `MessagesAreFromGroup` constraint is satisfied by checking if this query is empty. Similarly, the second query looks for the messages whose sent time is previous to its group creation time. Note that, although not explicitly stated, the joins and the projections in the previous queries are performed through the OID attributes of the tables.

We name this fragment of OCL as OCL\textsubscript{FO}\footnote{FO stands for First-Order, since relational algebra is, essentially, first-order logics.}. We define the syntax of OCL\textsubscript{FO} with a formal grammar and determine its semantics by means of set theory. In this way, the language is fully described in an unambiguous and concise way so that such concise description may be easily understood and adopted by current practitioners. Note that, those approaches aimed at bringing formal semantics to the full OCL require much larger definitions, and even relying on third party languages [4,12,13].

Regarding its expressiveness, we show that OCL\textsubscript{FO} is not only expressible in relational algebra, but equivalent to relational algebra. That is, every OCL\textsubscript{FO} constraint can be checked by means of a relational algebra query, and every constraint that can be checked by means of a relational algebra query can be written in OCL\textsubscript{FO}. This basic property ensures that OCL\textsubscript{FO} is as expressive as the main language of relational databases and guarantees that the complexity of checking the constraints is exactly the complexity of executing relational queries, that is, polynomial in data complexity (and in particular, AC\textsuperscript{0}). Moreover, it opens the door to reuse all the accumulated knowledge for efficient query answering in relational databases into efficiently checking of OCL\textsubscript{FO} constraints, as proposed by incremental approaches like the one in [14].

Finally, in order to make OCL\textsubscript{FO} an easy object of study, we also identify a core of the language that we call OCL\textsubscript{CORE}. Indeed, OCL\textsubscript{FO} is targeted to include the fragment of OCL that can be encoded into relational algebra. On the one hand, this makes OCL\textsubscript{FO} a language containing most of the operators that an OCL practitioner might use. On the other hand, this makes OCL\textsubscript{FO} a difficult object of study since it inherits a lot of the OCL syntactic sugar. The core of the language is aimed at overcoming this situation since it is a minimal fragment of OCL\textsubscript{FO} (consisting only of 6 operations) able to express any constraint written in the whole OCL\textsubscript{FO}.

The identification of this core of the language is also an important issue since it provides two significant contributions. First, it allows to easily state the relationship of any fragment of OCL with OCL\textsubscript{FO}, by determining whether this fragment incorporates or not the six operations in OCL\textsubscript{CORE}. Second, it entails that any implementation handling OCL\textsubscript{CORE} will also be able to deal with OCL\textsubscript{FO}.

This paper improves and extends our previous work in [15], where the syntax and the semantics of OCL\textsubscript{FO} were initially identified, in the following terms:

- We prove that checking general OCL constraints is not decidable (and not even semidecidable).
- We revisit the syntax and semantics of OCL\textsubscript{FO} to include the major part of OCL that can be encoded into relational algebra.
- We provide the full formal proof of the equivalence between OCL\textsubscript{FO} and RA.
- We introduce the concept of OCL\textsubscript{CORE}, a minimal subset of OCL\textsubscript{FO} with the same expressive power.

The paper is organized as follows. First, we define basic concepts in Section 2. Then, we show in Section 3 that checking general OCL constraints is not decidable. Afterwards, we present the language of OCL\textsubscript{FO} with its formal grammar and semantics in Section 4. In Sections 5 and 6 we prove that OCL\textsubscript{FO} constraints can be checked by relational algebra query emptiness, and viceversa. The OCL\textsubscript{CORE} is presented in Section 7. Finally, we review related work in Section 8, and discuss future work and conclusions in Section 9.

2 Basic concepts

UML class diagrams A UML Class diagram consists of a set of entity and relationship types. Entity types, also known as classes, have a name and a set of attributes; see for instance class `User` in our running example (Figure 1) with attributes `phone`, `state` and `lastConnect`. Relationship types, also called associations, are defined between pairs of entity types (we consider only binary relationship types); an example is `Writes`, defined between classes `User` and `Message`. Relationship types should not be confused with the so-called `is-a` relationship, which denote inheritance between classes; in our example, `Pair` and `Group` inherit from `ConversationGroup`, that is, they are subclasses of it, and it is superclass of them.

As we can see in Figure 1, UML class diagrams usually have graphical integrity constraints in the form of association cardinalities (e.g. `Writes` connects 1 author with 0 or more messages) or disjointness/completeness of entity type hierarchies (e.g. `Group` and `Pair` are disjoint, and each `ConversationGroup` is either a `Group` or a `Pair`). These are complemented with textual OCL constraints, which are the main focus of this paper.
An instance $I$ of a UML class diagram $S_{\text{UML}}$ is a finite set of objects (instances of classes) and links (instances of associations). An object is identified by an OID and has a value for each of the class’ attributes. A link is a pair relating the OIDs of the connected objects. We say that $I$ is consistent if it satisfies all constraints in $S_{\text{UML}}$.

**OCL constraint** An OCL constraint is a textual expression, attached to a UML class diagram, stating a condition that any UML instance for that schema should always satisfy. OCL constraints, as shown in Figure 2, are defined by means of a context, an invariant name, and a boolean expression.

The boolean expression of the OCL constraint should be satisfied for any given instance of the context class (referred to as ‘self’). That is, given a UML instance, the OCL top boolean expression of the constraint should evaluate to true when interpreting self as any UML object of the given context class.

OCL offers several operators for building the OCL top boolean expression, where the most basic one is the navigation operation (referred to as ‘.’). Intuitively, a navigation from an object (set of objects) to some property, is an operation that, given the initial object (set of objects) collects the values/objects related to it according to such property. For instance, the constraint MessagesAreFromGroup from Figure 2 has the navigation self.msg. This navigation obtains the messages related to the group self. In addition, navigations can be naturally concatenated. Following with our example, self.msg.author obtains the users who are the authors of the messages related to self.

Given one or several OCL navigations, the OCL language provides several operators to check them such that it is possible to return a boolean value. For instance, in MessagesAreFromGroup, the OCL operator includesAll checks whether the contents of the navigation of self.msg.author is contained in the navigation self.user.

**Relational schemas** A relational schema is a set of relations $\{R_1, \ldots, R_n\}$, each with a particular signature $R(A_1, \ldots, A_m)$, where $A_1, \ldots, A_m$ are the attributes, and $R$ is the relational symbol (with arity $m$). We usually identify it with its relational symbol.

An instance $D$ of a relational schema $S_{\text{REL}}$ is a set of tuples in the form of $R(a_1, \ldots, a_m)$, where $R$ is a relational symbol from $S_{\text{REL}}$ with arity $n$, and $a_1, \ldots, a_m$ are constants.

**Relational views of UML class diagrams** In this paper, we consider a particular class of relational schemas that are derived from UML class diagrams. We say that $S_{\text{REL}}$ is the relational view of UML class diagram $S_{\text{UML}}$ if it is a relational schema constructed by applying the following rules, starting from an empty set of relations:

- For each class $C$ in $S_{\text{UML}}$ with attributes $\text{attr}_1, \ldots, \text{attr}_n$, add $n$-ary relation $C$ to $S_{\text{REL}}$ with signature $C(\text{oid}, \text{attr}_1, \ldots, \text{attr}_n)$.

- For each association $A$ in $S_{\text{UML}}$ between any pair of classes $C_1$ and $C_2$, add binary relation $A$ with signature $A(c_1, c_2)$ to $S_{\text{REL}}$.

Note that, assuming a UML class with $n$ attributes, its relational translation is indeed a $n$-ary relation due to the addition of the OID. The concept of relational view is easy to reformulate at the level of instances if the OID of each tuple is set to a fresh value, different from the OID of any other tuple in the whole instance. The actual value of the OID is irrelevant, all that matters is its uniqueness with respect the OIDs of the other tuples. For the sake of simplicity, we assume single-valued, primitive-typed attributes.

More formally, let $S_{\text{REL}}$ be the relational view of $S_{\text{UML}}$, and $D$ and $I$ instances of $S_{\text{REL}}$ and $S_{\text{UML}}$, respectively. We say that $D$ is the relational view of $I$ if it is constructed from the empty instance, using the following rules:

- For each class $C \in S_{\text{UML}}$ and each object $c \in I$ of $C$, where $v_1, \ldots, v_n$ are the attribute values of $c$, add tuple $C(c, v_1, \ldots, v_n)$ to $D$.

- For each association $A \in S_{\text{UML}}$ and each link $a = (c_1, c_2) \in I$ instance of $A$, add tuple $A(c_1, c_2)$ to $D$.

A relational view $D$ of a UML instance $I$ is consistent if $I$ is consistent.

Regarding graphical constraints, including generalization set constraints such as incomplete or disjoint, and their translation into the relational view, providing a specific translation for them would be redundant, as they can be expressed in OCL and translated into RA with the algorithms shown later in the paper. The unique exception are $\text{IsA}$ hierarchy constraints, which can be encoded using relational foreign keys (subclass relations pointing to superclass relations).

Other UML aspects intended for software design rather than software specification (such as interfaces and directed associations) are outside of OCL's framework, and thus, outside of the scope of this article.

We use $R_{\text{VIEWS}}$ to denote the class of relational views of UML class diagrams, i.e., $R_{\text{VIEWS}} = \{S_{\text{REL}} \mid \exists S_{\text{UML}} : S_{\text{REL}}$ is the relational view of $S_{\text{UML}}\}$. Henceforth, whenever we say “relational schema”, we mean a schema that belongs to $R_{\text{VIEWS}}$.

**Substitutions** A substitution is a function that, when applied to an OCLFO expression, replaces free variables
with constants. We use \( E_{[s]} \) to denote the application of a substitution \( s = \{ \text{var}_1 \to c_1, \text{var}_2 \to c_2, \ldots \} \) to an expression \( E \).

A substitution \( S_t \) obtained from some tuple \( t \) is a substitution that replaces variables with the values appearing in \( t \). By default, \( S_t \) replaces a variable \( \text{var} \) with the value for the relational attribute called \( \text{var} \) appearing in \( t \), i.e., \( t[\text{var}] \). Conversely, we can also define a mapping \( M \) from OCL variables to relational attributes in \( t \), so that, \( S_t \) replaces a variable \( \text{var} \) with \( t[M(\text{var})] \).

Similarly, we say that a substitution \( S_t \) is obtained from some query \( q \) on some instance \( I \) when there exists some tuple \( t \in q(I) \) from which \( S_t \) is obtained.

3 Undecidability of OCL Constraint Checking

It is well known that there is a tradeoff between the expressiveness of a language and the computational complexity to reason with it. I.e., the greater its expressiveness, the more difficult is to evaluate/analyze its expressions.

Bearing this in mind, we are interested to know the computational complexity of checking a general OCL constraint in terms of data complexity. That is, how much difficult is to evaluate an OCL constraint regarding the size of a UML instantiation. Surprisingly, to our knowledge, although there are similar studies on other problems like OCL maintenance [16] or reasoning [17], this analysis has not been yet performed for integrity checking of OCL constraints.

In fact, it turns out that, unfortunately, current full OCL is so expressive that it is not decidable. That is, it is impossible to build an algorithm guaranteed to terminate and correctly assessing whether an OCL constraint is satisfied or not in an arbitrary UML instantiation. Things get even worse because we can also prove that OCL is not even semidecidable. Therefore, the algorithm is not ensured to terminate even in the case of UML instantiations that satisfy the constraint. Clearly, this entails a huge problem to OCL constraint evaluator techniques. Indeed, this result implies that checking techniques might hang when evaluating if some valid UML instantiation is valid, not to say that decidable cases might take an exponential amount of time.

We prove the previous results by means of reducing a non-decidable problem into OCL constraints checking. In particular, we reduce the problem of checking whether some word is accepted by a type-0 grammar [18] into OCL checking. Roughly speaking, a type-0 grammar is a set of replacing rules that transforms a finite string into another finite string. Thus, the word-acceptance problem in such grammar consists in, given some word, checking whether this word is generated by the set of production rules and some initial symbol. It is known that such problem is undecidable since, roughly speaking, a type-0 grammar can emulate a turing machine.

The reduction is based on exploiting OCL recursion, and OCL string operators. The idea is to build an OCL operation \( \text{produces} \) that recursively checks if a given word \( \text{target} \) is generated from an initial word \( \text{current} \) by means of a set of production rules. This is done by replacing the initial word into another one by using OCL String concatenation and substring operators. If the new word coincides with the \( \text{target} \), the operation returns true, otherwise, there is a recursive call to check if transforming the new word we can generate the \( \text{target} \).

The key of the undecidability relies on the finite, but unbounded, memory consumption required for such operation. Indeed, our OCL operation rewrites finite strings into new finite strings, but the size of the new strings cannot be bounded, which makes the recursion potentially infinite. This behavior mimics the one of the type-0 grammars (or turing machines). A Turing Machine might hang because of never reaching the final state consuming more and more tape (although it is a finite amount of tape). Equivalently, the type-0 grammar might derive, at runtime, non-terminal Strings of greater size each time, without never reaching the terminal word. This runtime derivation of non-terminal words of (potentially) increasing size is achieved in our setting through the recursive \( \text{produces} \) OCL operation, whose \( \text{current} \) input String might become longer and longer.

In the following, we formally state and proof such result.

**Theorem 1** Checking whether an OCL constraint is satisfied in an arbitrary UML instantiation is not decidable, and not even semidecidable.

**Proof** First, we prove non-decidability. Then, we prove non-semidecidability.

To prove non-decidability, we make a reduction from a non-decidable problem to the problem of OCL integrity constraints checking. Thus, the undecidability of the former implies the undecidability of the latter. In particular, we make the reduction from the problem of word acceptance in a type-0 grammar. A type-0 grammar is a kind of grammar where all symbols (terminal, and non-terminal) might be substituted by means of the grammar rules. A formal definition of type-0 grammar might be found in [18], but, for our purposes, we only need to know that checking if a word is produced (aka accepted) by means of a type-0 grammar is undecidable.
Bearing this in mind, our proof strategy consists in building an OCL constraint stating that a word is accepted, if and only if, it is produced by the grammar. To do so, we define a UML class diagram encoding type-0 grammars on one side, and accepted/non accepted words on the other, so, we can define an OCL constraint imposing our intended constraint between the two.

In particular, our UML class diagram, shown in Figure 3, is capable of encoding all the production rules of an arbitrary type-0 grammar. A production rule is composed of a left and right hand sides, where each side is a sequence of symbols. These symbols might be classified into non-terminal, and terminal, where non-terminal symbols might be further classified into start symbols. Moreover, the UML class diagram also contains the notion of word, where words might be classified into accepted/non-accepted words. Note that, any type-0 grammar and accepted/non-accepted word can be described through this UML class diagram, i.e., any type-0 grammar and word can be written as an instance of this UML class diagram.

```
context ProductionRule def produces (current: String, target: String): Boolean = self.replacements(current)->exists(newWord
  newWord = target or
  ProductionRule.allInstances()->exists(p
    p.produces(newWord, target))

This definition makes use of the replacements operation, which returns the different words we can obtain by applying the production rule. Replacements can be defined recursively in OCL. Intuitively, we need to first compute the strings representing the left and right part of the production rule. Then, if the input word is empty, there is no replacement to apply and thus, we can return the empty set. Otherwise, we have to compute: 1) any replacement that can be performed from the first character, and 2) all the replacements that can be performed from the second character and beyond. We can compute 1) by checking if the word's beginning matches the left part of the rule, and if so, add into the result the word corresponding to replace the beginning of the current word with the right part of the production rule. 2) can be computed recursively by means of skipping the first character, and then, iterate the given result to concatenate this first character at the beginning of each returned word. Formally:

```
context ProductionRule def replacements (current: String): Set(String) =
  let leftW: String = self.left
  -iterate(s;1: String = "" | l+s.symbol) in
  let rightW: String = self.right
  -iterate(s;r: String = "" | r+s.symbol) in
  if current = "" then Set()
  else if current.substring(1,leftW.size())=leftW
  then Set(rightW+
    current.substring(leftW.size(),
    current.size()))
  else Set() endif
  -union(
    self.replacements{
      current.substring(2,current.size())
    }->iterate(i; ac: Set{})
    ac->including(current.substring(1,1+i))
  )
endif
```

Thus, we can check whether some word is accepted by a type-0 grammar by instantiating the word and the grammar in the previous UML class diagram and checking the satisfaction of the given IsProducedWord OCL constraint. Since checking whether some word is accepted by a type-0 grammar is not decidable, check-
ing OCL constraints is not decidable either. This concludes the first part of the proof.

Now, to prove that checking OCL constraints is not even semidecidable, we reduce the problem of checking whether some word is rejected by some type-0 grammar, which is a well known non-semidecidable problem.

Let us consider a constraint specifying that non-accepted words are words that cannot be produced by the grammar. In OCL, this constraint can be stated by simply negating the previous one:

```ocl
class NonAccepted
  inv IsNotProducedWord:
    not ProductionRule.allInstances().exists(p | StartSymbol.allInstances().exists(s | p.produces(s, s.word, s.word)))
```

We can check now whether some word is rejected by a type-0 grammar by instantiating the word and the grammar in the previous UML class diagram and checking the satisfaction of the given `IsNotProducedWord` OCL constraint. Therefore, checking OCL constraints is not even semidecidable.

After defining the syntax and the semantics of OCLFO, we make a brief discussion about the OCL operations outside OCLFO while distinguishing whether they could be effectively emulated in OCLFO, or not. Then, we conclude this section analyzing the coverage of OCLFO with two UML/OCL case studies.

4 OCLFO fragment of OCL

We provide in this section the syntax and the semantics of OCLFO. Since our goal is to use OCLFO as a language for specifying constraints, we place special emphasis on OCLFO boolean statements.

The syntax is defined through a formal grammar which limits the standard OCL boolean statements to those that can be computed through RA queries. Since, intuitively, relational algebra is known to be equivalent to (domain independent) first-order logic, such grammar leaves out the OCL higher-order operators like transitive closure, or (most) aggregation functions.

Semantics is defined by means of set theory. Roughly, OCL navigations are interpreted as sets, or single objects/values, and OCL boolean operators are interpreted as checks over them (e.g. the semantics of `includes` consists in checking whether some object/value belongs to a set, etc.). Therefore, this semantics does not distinguish among different collection types, as OCL does (e.g. it does not contemplate bags, nor ordered sets). However, this is not a drawback of OCLFO but a necessary limitation to ensure equivalence with RA (since RA only supports sets). Furthermore, most of the OCLFO considered operators regarding collections brings the same results when interpreting an OCL collection as a set. In fact, the unique ones that might differ are: `size`, `one`, and `isUnique` in case that its `collection source` included a navigation that might contain repeated objects.

4.1 OCLFO Syntax

The grammar of OCLFO is stated in Figure 4. Briefly, an OCLFO constraint is an OCL-Bool statement written in some class context as shown in Figure 2. Such boolean statement might make use of `navigations`, i.e., OCL-Set, OCL-Object, or OCL-Value statements. Intuitively, the first kind of statements describe a set of objects, whereas the last two determine a single object/value, respectively. Such navigations are then used as the input of some OCLFO operator to obtain the OCL-Bool statement that defines the constraint.

These OCLFO statements are built over a `signature` consisting of a set of class names, role/attribute names (where some might be functional, i.e., with a maximum cardinality of 1), association class names, and constant names. Typically, this signature is provided by an associated UML class diagram.

For the sake of simplifying the language, we limit OCLFO statements to evaluate to valid results. Thus, we require the expressions to apply the proper safety checks to avoid raising the OCL `invalid` value in runtime. For instance, if we have some object of type `T1`, and we want to cast it to `T2`, we might need to check that the object has also the type `T2` (unless `T1` is a subclass of `T2`). Note that we can analyze in which cases these safety checks necessary with a syntactic inspection of the OCL expression and the class diagram.

4.2 OCLFO semantics

To define the semantics of OCLFO we interpret the OCL-Bool statements as `true/false` values, the OCL-Set statements as sets of objects/values, and the OCL-Object/OCL-Value statements as a single object/value respectively.

We define first the semantics of OCLFO without considering `null` values. That is, assuming that when interpreting an OCL-Object or OCL-Value expression, we always reach some defined object/value. We will include the treatment of nulls later on.

4.2.1 OCLFO semantics without NULLS

The semantics of an OCLFO statement is defined through the interpretation of its signature. Such inter-
OCL-Bool ::= OCL-Bool BoolOp OCL-Bool | not OCL-Bool |
   OCL-Set ->includesAll(OCL-Set) | OCL-Set ->excludesAll(OCL-Set) |
   OCL-Set ->includes(OCL-Single) | OCL-Set ->excludes(OCL-Single) |
   OCL-Set ->forall(VarList | OCL-Bool) |
   OCL-Set ->isEmpty() | OCL-Set ->notEmpty() |
   OCL-Set ->size() CompOp Integer | ->one(Var | OCL-Bool) |
   OCL-Set ->isUnique(attr) |
   OCL-Object.oCompletionOf(Class) | OCL-Object.oIsTypeOf(Class) |
   OCL-Object = null | OCL-Object <> null |
   OCL-Value CompOp OCL-Value |
   OCL-Object.bAttr | Var

OCL-Navigation ::= OCL-Set | OCL-Single

OCL-Set ::= OCL-Set ->union(OCL-Set) | OCL-Set ->intersection(OCL-Set) |
   OCL-Set ->symmetricDifference(OCL-Set) | OCL-Set = OCL-Set |
   OCL-Set ->select(Var | OCL-Bool) | OCL-Set ->project(Var | OCL-Bool) |
   OCL-Set ->selectByKind(Class) | OCL-Set ->selectByType(Class) |
   OCL-Set.role | [role] | OCL-Set.asClass | [role] |
   OCL-Object.nRole | [role] | OCL-Object.nIsAsClass | [role] |
   OCL-Set.attr | OCL-Object.nAttr |
   Class.allInstances() | OCL-Single |

OCL-Single ::= OCL-Object | OCL-Value

OCL-Object ::= OCL-Object.oCompletionOf(Class) |
   OCL-Object.fAsClass | Var | self

OCL-Value ::= Constant | Var |
   OCL-Object.fAttr |
   OCL-Set ->minO | OCL-Set ->maxO |

BoolOp ::= and | or | xor | implies

CompOp ::= < | <= | = | >= | > | <>

VarList ::= Var | Var* |

Var ::= (a variable name)

Class ::= (a class name)

assocClass ::= (an association class name)

fAssoClass ::= (an association class name of a functional role)

nfaClass ::= (an association class name of a non-functional role)

role ::= (a role name)

fRole ::= (a functional role name)

nrole ::= (a non-functional role name)

attr ::= (an attribute name)

bAttr ::= (a boolean attribute name)

fAttr ::= (a functional attribute name)

nfaAttr ::= (a non-functional attribute name)

Integer ::= (an integer number)

Constant ::= (a constant name)

Fig. 4. Syntax of OCLFO

Formally, an interpretation is a pair \( I = (\Delta^I, \mathcal{I}) \), where \( \Delta^I \) is a non-empty set of object identifiers and values referred as the interpretation domain, and \( \mathcal{I} \) is a function, referred as interpretation function, that maps each element in the signature of the OCL-FO statements to \( \Delta^I \) tuples. In particular, class names are mapped to a set of domain elements, attribute names are mapped to a set of domain element pairs, k-ary association (class) names are mapped to a set of k-ary (k+1-ary) domain element tuples, and constant names are interpreted to domain elements with the same name (i.e., we follow the standard name assumption).
Fig. 5 Semantics of $\text{OCL}_{10}$
For example, an interpretation $I_0$ for the signature defined by our running UML class diagram example might be:

$$
\Delta^{I_0} = \{ \#user1, \#user2, \#group1, \#msg1, \\
1/1/2016, 12/12/2015, 'Happy new year!', \ldots \}
$$

$$
\text{User}^{I_0} = \{ \#user1, \#user2 \}
$$

$$
\text{Group}^{I_0} = \{ \#group1 \}
$$

$$
\text{Message}^{I_0} = \{ \#msg1 \}
$$

$$
\text{crTime}^{I_0} = \{ < \#group1, 12/12/2015 > \}
$$

$$
\text{sentTime}^{I_0} = \{ < \#msg1, 1/1/2016 > \}
$$

$$
\text{isSentTo}^{I_0} = \{ < \#msg1, \#group1 > \}
$$

Given an interpretation $I$, an OCL$_{FO}$ statement is interpreted according to the recursive definition specified in Figure 5. Such definition is mainly provided in terms of set theory, together with some relational algebra operators (such as join $\Join$, project $\pi$, or select $\sigma$), to easily define the interpretation of OCL$_{FO}$ navigations. Moreover, to define these navigations, we assume that $ns$ is the role name of the navigation source which corresponds to the [role] expression of a navigation, or the opposite of some role in navigations through binary associations. Lastly, we use the expression $\text{Subclasses(Class)}$ to refer to those objects belonging to a subclass of $\text{Class}$.

Thus, given an OCL$_{FO}$ constraint $\phi$ of the form:

\[
\textbf{context R inv ConstraintName: OCL-Bool}
\]

We say that an interpretation $I$ satisfies the constraint $\phi$, and write $I \models \phi$ if and only if it evaluates to TRUE for all the objects of its context class $R$. More formally:

$$
I \models \phi \iff \forall v \in R, \text{OCL-Bool}^I_{\phi_{def}} = \text{TRUE}
$$

For example, $I_0$ satisfies the OCL$_{FO}$ constraint MessagesAreSentAfterCreation since for its unique group $\#group1$ it holds that:

$$
\{ v \in \pi_{msg1}\{\#group1\} \times \text{isSentTo}^{I_0} \mid \\
\pi_{sentTime}\{\{v\} \times \text{sentTime}^{I_0} \times v < \\
\pi_{crTime}\{\{\#group1\} \times \text{crTime}^{I_0} \} \}
$$

In case $I$ satisfies $\phi$, we say that $I$ is a model of $\phi$, otherwise, we say that $I$ violates $\phi$. We naturally extend the notions of model, satisfaction, and violation to sets of OCL$_{FO}$ constraints $\Phi$. For instance, $I$ satisfies a set of constraints $\Phi$ if and only if $I$ satisfies each $\phi \in \Phi$.

### 4.2.2 OCL$_{FO}$ semantics with nulls

Sometimes, the interpretation of some OCL-Object or OCL-Value results into no value. Indeed, consider that in our running example, some message $\#msg1$ has no value defined for the attribute $\text{sentTime}$. In this case, when navigating from the user $\#msg1$ to its $\text{sentTime}$, we obtain no value. More formally, we obtain $\emptyset$.

In such case, we define the OCL-Object/OCL-Value to be interpreted as a new value called NULL not present in $\Delta^I$. In particular, we cast the $\emptyset$ to NULL (and vice versa) depending on the OCL expression it appears. Note that such interpretation corresponds to the one given in the OCL standard in [4].

Thus, when $\emptyset$ appears when interpreting some OCL-Object/OCL-Value, we automatically cast it to the new value NULL. Since the NULL value is not present in $\Delta^I$, the NULL value does not join any value in the signature interpretation $I$. That is, it does not join any value present in the interpretation of any association or attribute. This implies that, when navigating from a NULL value to obtain another object/value, we obtain again $\emptyset$, which is cast to NULL if this navigation is an OCL-Object/OCL-Value. This behavior perfectly emulates the standard OCL semantics proposal [4].

Finally, we need to extend the interpretation of the OCL-Bool to determine if they are evaluated to TRUE or FALSE when they use some OCL-Object/OCL-Value that evaluates to NULL. This differs from standard OCL since OCL considers that an OCL-Bool expression might return NULL or even INVALID. However, since we need OCL$_{FO}$ to be a two-valued logic language (as it is first-order logics), we restrict OCL-Bool values to either TRUE or FALSE.

Thus, and following the criteria already used in [19], we consider that an OCL-Bool is true if some of its OCL-Object/OCL-Value subexpression are evaluated to NULL. The idea behind this interpretation is that an OCL$_{FO}$ constraint is not violated unless the values that determine its satisfaction/ violation are defined. For instance, the constraint MessagesAreSentAfterCreation would be satisfied in the previous example if $\#msg1$ had not $\text{sentTime}$ defined yet, since then, its sent time would not be previous to the creation group time.

Note that this interpretation is just a default behavior to apply in case of finding a NULL value. However, note that it is possible to write a constraint that it is violated when some of its expressions evaluate to NULL by simply adding the boolean subexpression and OCL-Single/OCL-Object $\not\rightarrow$ null to the OCL constraint.

### 4.3 OCL Operations not in OCL$_{FO}$

There are two kinds of OCL operations that are not inside OCL$_{FO}$. On one hand, we have operations that are outside the expressiveness of OCL$_{FO}$, and on the
other, operations which we have intentionally left out for the sake of simplicity.

As we will prove in the next two sections, $\text{OCL}_{FO}$ is expressively equivalent to relational algebra, and therefore, equivalent to (domain independent) first-order logic. Thus, it can be stated straightforwardly that all OCL operations which are not first-order (or not inside relational algebra) cannot be included in $\text{OCL}_{FO}$.

Examples of operations that are not first-order are closure, or aggregation functions such as count. It is worth noting, however, that the aggregation function size can be used in $\text{OCL}_{FO}$ to compare the size of some set with some fixed integer, but not with another set size neither with the value obtained from some attribute. Examples of operations that are outside because of relational algebra do not support them are basic type operations such as $\oplus$, $\ominus$, $\ast$, and $\div$. In this last case, however, we argue that these operations could be integrated into relational algebra (e.g. in the selection operation), and thus, could be integrated into $\text{OCL}_{FO}$.

For the sake of simplicity, we have left out operations like let in, if then else endif, including or excluding. This has been done for the sake of having a compact fragment of the language. We argue that these operations that we have left out are not frequently used when defining OCL constraints. Indeed, none of them appear, for instance, when defining constraints in the UML class diagrams of the DBLP case study [20] and the osCommerce UML/OCL case study [21], the case studies we have used to check the $\text{OCL}_{FO}$ coverage.

In any case, we would like to highlight that, since $\text{OCL}_{FO}$ captures relational algebra, it is as expressive and useful for defining constraints as, in essence, standard SQL assertions [22].

### 4.4 $\text{OCL}_{FO}$ coverage evaluation through case studies

In this section, we discuss the expressiveness coverage of the $\text{OCL}_{FO}$ fragment for defining constraints. By coverage, we mean the ratio of constraints that can be written in $\text{OCL}_{FO}$ in comparison to full OCL, when considering conceptual schemas for real systems.

In the following, we first discuss the selected case studies (DBLP [20], and osCommerce [21]), and then, show and discuss the $\text{OCL}_{FO}$ coverage.

#### 4.4.1 The DBLP and osCommerce case studies

The DBLP and the osCommerce systems are two UML/OCL conceptual schemas describing real systems that have been used several times as case studies in conceptual modeling research.

The DBLP case study is a conceptual schema describing the DBLP system, the service that provides open bibliographic information on major computer science journals and proceedings. This UML/OCL conceptual schema consists of 17 classes, including concepts such as Publication, ConferenceEdition, Journal, etc.

The osCommerce case study is a conceptual schema describing the osCommerce platform, an open-source online store management program. This UML/OCL conceptual schema consists of around 40 classes, including concepts such as Product, Order, PaymentMethod, etc.

#### 4.4.2 Evaluation of $\text{OCL}_{FO}$

In the case of DBLP, 25 out of the 26 OCL constraints were inside the $\text{OCL}_{FO}$ fragment. The unique constraint that remained outside was one speaking about Sequences which is a collection type we do not consider in $\text{OCL}_{FO}$ since relational algebra works with unsorted collections.

In the case of osCommerce, 30 out of 33 OCL constraints were inside the $\text{OCL}_{FO}$ fragment. Regarding the other 3, one was encodable in $\text{OCL}_{FO}$ after rewriting (it was using an OCL Tuple construct that could be easily removed). The other two were constraints about transitive closure and collection size comparisons, which are outside first-order logics and thus, outside $\text{OCL}_{FO}$.

Thus, from a total of 59 constraints extracted from real systems, we see that $\text{OCL}_{FO}$ could encode directly 55, and could encode 56 after considering some rewriting. This represents a coverage of around 95%, which, in our opinion, makes the $\text{OCL}_{FO}$ language useful in actual cases since its expressiveness allows to cover most of the constraints that are required in practice.

### 5 Checking $\text{OCL}_{FO}$ Constraints by RA Queries

The main goal of this section is to show that any $\text{OCL}_{FO}$ constraint can be checked by means of a RA query since given an $\text{OCL}_{FO}$ constraint we can build a RA query that retrieves the objects that violate it. Therefore, the $\text{OCL}_{FO}$ constraint is satisfied if and only if its corresponding query is empty.

This result entails that checking an $\text{OCL}_{FO}$ constraint is at most as difficult as executing a RA query and, thus, checking an $\text{OCL}_{FO}$ constraint can be solved in polynomial time with regards to data complexity (and in particular, it belongs to the $\text{AC}^0$ complexity class). Moreover, this also enables reusing current
techniques developed in the community of relational databases for the treatment of OCL\_FO constraints. For instance, incremental OCL\_FO constraints checking can be solved by means of incremental query answering (e.g., following [14], or [23]), and repairing OCL\_FO constraints can be solved by view updating techniques (as studied for instance in [16]).

The RA query is built from an OCL\_FO constraint in two steps. First, we normalize the OCL\_FO constraint into an equivalent one that uses a lower number of OCL\_FO operators. Then, we translate the normalized OCL\_FO constraint into a RA query that returns the objects that violate the constraint.

5.1 Normalizing OCL\_FO Constraints

The grammar presented in Figure 4 admits very complex OCL expressions, using lots of different OCL operations. Therefore, defining directly a translation from pure OCL\_FO constraints into RA queries becomes very cumbersome. To make things simpler, we initially translate each OCL\_FO constraint into a normalized one, whose expression is defined only by means of a small number of operations.

We say that an OCL\_FO constraint is normalized if it is defined only by means of the following operations: and, not, forAll, =, and < for OCL\_Bool expressions; and union, intersection, -, select for OCL\_Set expressions; although their expression can also contain the usual navigations through roles and attributes, and the oclAsType cast operation.

To normalize an OCL\_FO constraint, we recursively apply the rewritings defined in Figure 6. It is not difficult to verify that such rewriting preserve the original semantics of the constraint by means of directly inspecting the operation semantics defined in Figure 5.

In our running example, the normalized version of the constraint MessagesAreFromGroup is:

```
self.msg.author->forAll(a
    not self.user->forAll(u|not a = u))
```

Note that the fragment of OCL obtained after normalization is only of internal use to facilitate the translation of OCL\_FO to RA queries. So, we do not expect the designer to make use of this fragment when specifying integrity constraints since constraints in OCL\_FO can be automatically normalized as stated before.

5.2 Drawing RA Queries from Normalized Constraints

To obtain the RA query of a normalized OCL\_FO constraint, we first translate each OCL\_FO navigation (i.e., OCL\_Set, OCL\_Object and OCL\_Value expressions) into RA queries retrieving its corresponding set, object, or value. Then, the resulting RA query is obtained by translating the OCL\_Bool expressions through the composition of the translation of its navigations.

More precisely, the crucial point for translating the navigations are the OCL\_FO variables (i.e., the self variable and the iteration variables appearing in forAll and select expressions) since our goal is to build a RA query with one relational attribute for each OCL\_FO variable alive in the OCL\_FO navigation, together with one additional attribute containing the result of the navigation. For instance, when translating the normalized MessagesAreFromGroup constraint, the navigation self.msg.author is translated as a relational query with the attributes self and result.

Intuitively, when executing such queries over some interpretation I, each retrieved tuple t represents a combination of values that the OCL\_FO variables may take when evaluating the OCL\_FO navigation with I. For instance, if in the interpretation I we have that a group #group1 has some message #msg1 written by #John, then, the row <#group1, #John> appears in the query result that translates the navigation self.msg.author.

Then, the idea of composing those translations to obtain the translation of the whole OCL\_Bool expression defining the constraint is to select those tuples that witness the violation of the boolean condition. That is, we want to select the row <#group1, #John> from the previous example in case that #John is not a member of #group1.

All translations are recursive and use the input variable q\_c, the context query, which is the relational query that retrieves the values for the alive OCL\_FO variables defined in the upper expression of the expression being translated. For instance, to translate self.msg.author, we need a context query q\_c retrieving the values that the variable self might take, e.g., q\_c = Group.

In the rest of this section, we first present the algorithms for translating each OCL\_FO navigation, while discussing their intuition and providing a formal proof of their correctness. Afterwards, we show and prove how to use these algorithms to translate the whole OCL\_Bool expression defining the OCL\_FO constraint.

For the seek of simplicity, we omit some relational algebra low-level details such as relational algebra attribute renamings and some selection conditions since they can be easily understood from the context.
5.2.1 OCL-Set Translation

Algorithm 1 translates an OCL-Set into a relational query retrieving the same values/objects as the ones in the OCL-Set. Since the OCL-Set expression might have OCL free variables (such as self), we need the context query qC to bring the different possible substitutions to apply to such variables. Thus, each tuple t in the result of the query has the form <t1, ..., tn, v>, where t1, ..., tn, represents a substitution for the OCL\(_{\text{FO}}\) variables appearing in the OCL-Set (which are taken from qt, and v a value appearing in the OCL-Set according to such substitution for the free variables.

The idea behind the algorithm is to use the relational operation corresponding to each OCL\(_{\text{FO}}\) set operation. I.e., union is translated into \(\cup\), role navigations as \(\land\), etc. The major difficult, however, relies on the translation of the select operation, which is translated using the translation of OCL-bool expressions. In this case, the idea is to first, translate the OCL-Set source expression, and then, remove all those rows not satisfying the inner OCL-bool expression.

As an example, consider the OCL-Set expression self\_msg\_author with a context query \(q_c=\text{Group}\) defining the values that the \(\text{OCL}_{\text{FO}}\) variable self might take. Such expressions is translated as:

\[
\pi_{\text{group,author}}(\text{Group} \times \text{IsSentTo} \times \text{Writes})
\]

Intuitively, the translation just translates the role navigations to RA joins, and then, projects the result to only retrieve the reachable messages for each group.

In the following we formally prove the correctness of this algorithm.

Theorem 2 Let \(\phi\) be an OCL-Set over a UML class diagram \(\mathcal{S}_{\text{UML}}\), \(\overline{\phi}\) a relational algebra expression defined over the relational view of \(\mathcal{S}_{\text{UML}}\), and \(q_c\) a context query such that \(\overline{\phi} = r\text{Translation}(\phi, q_c)\) (Algorithm 1). Then,

\[
v \in q_c[S_i] \text{ iff } <t[0],...,t[n],v> \in \overline{\phi}(I)
\]

for any value \(v\), any interpretation \(I\), and any substitution \(S_i\) obtained from \(q_c[I]\).

Proof The proof is based on structural induction over the OCL\(_{\text{FO}}\) grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. Consider the base case:

\[
\phi = \text{R.allInstances()}
\]
Algorithm 1 raTranslation(OCL-Set, q_e)

if OCL-Set = OCL-Set_1 ∪ ¬union(OCL-Set_2) then
    q_1 := raTranslation(OCL-Set_1, q_e)
    q_2 := raTranslation(OCL-Set_2, q_e)
    return q_1 \cup q_2
else if OCL-Set = OCL-Set_1 \ intersection(OCL-Set_2) then
    q_1 := raTranslation(OCL-Set_1, q_e)
    q_2 := raTranslation(OCL-Set_2, q_e)
    return q_1 \cap q_2
else if OCL-Set = OCL-Set_1 \setminus OCL-Set_2 then
    q_1 := raTranslation(OCL-Set_1, q_e)
    q_2 := raTranslation(OCL-Set_2, q_e)
    return q_1 \setminus q_2
else if OCL-Set = OCL-Set_1 \selectFor(OCL-Bool) then
    q_1 := raTranslation(OCL-Set_1, q_e)
    return q_1 \setminus q_r
else if OCL-Set = OCL-Set_1 \ role then
    q_1 := raTranslation(OCL-Set_1, q_e)
    return π(q_1 \ \\ M \ am(role))
else if OCL-Set = OCL-Set_1 \ assocClass then
    q_1 := raTranslation(OCL-Set_1, q_e)
    return π(q_1 \ \\ M \ assocClass)
else if OCL-Set = OCL-Object_1 \sizeofthen then
    q_1 := raTranslation(OCL-Object_1, q_e)
    return π(q_1 \ \\ M \ sizeofthen)
else if OCL-Set = OCL-Object_1 \ assocClass then
    q_1 := raTranslation(OCL-Object_1, q_e)
    return π(q_1 \ \\ M \ assocClass)
else if OCL-Set = OCL-Set_1 \ attr then
    q_1 := raTranslation(OCL-Set_1, q_e)
    return π(q_1 \ \\ M \ attr)
else if OCL-Set = OCL-Object_1 \ attr then
    q_1 := raTranslation(OCL-Object_1, q_e)
    return π(q_1 \ \\ M \ attr)
else if OCL-Set = R.allInstances() then
    return q_1 \times R
else if OCL-Set = OCL-Object_1 then
    return raTranslation(OCL-Object_1, q_e)
end if

According to Algorithm 1,

$$\overline{q} = \text{raTranslation}(\phi, q_e) = q_e \times R$$

Clearly, according to the semantics of OCL_{FO}, \( v \in R.allInstances() \) if \( v \in R \). Moreover, it is guaranteed that \( S_1 \) is obtained from \( q_e(I) \). Hence, \( v \in R.allInstances() \) if \( v <\ll 0, ..., n] \), \( v > \in q_e(I) \times R \).

Consider the inductive case:

$$\phi = \text{OCL-Set} \cdot \select(s \mid \text{OCL-Bool})$$

According to Algorithm 1,

$$\overline{q} = \text{raTranslation}(\phi, q_e) = q_s \setminus q_r$$

where

$$q_s = \text{raTranslation}(\text{OCL-Set}, q_e)$$
$$q_r = \text{raTranslation}(\text{OCL-Bool}, q_e)$$

According to the semantics, we have that \( v \in \text{OCL-Set} \cdot \select(s \mid \text{OCL-Bool}) \) if and only if \( v \in \text{OCL-Bool} \cdot \text{OCL-Set} \). This is the case if and only if \( v \in \text{OCL-Set} \cdot \select(s \mid \text{OCL-Bool}) \) is true. Equivalently, this is the case if \( v \in \sum_{s} \left[ S_{\text{OCL}} \right] \) and not \( \text{OCL-Bool} \cdot \sum_{s} \left[ S_{\text{OCL}} \right] \) is false.

Now, by induction hypothesis, we have that \( v \in \text{OCL-Set} \cdot \select(s \mid \text{OCL-Bool}) \) if \( v <\ll 0, ..., n] \), \( v > \in q_e(I) \), and \( \text{OCL-Bool} \cdot \sum_{s} \left[ S_{\text{OCL}} \right] \) is false if \( v <\ll 0, ..., n] \), \( v > \in q_e(I) \). Hence, \( v \in \text{OCL-Set} \cdot \select(s \mid \text{OCL-Bool}) \) if \( v <\ll 0, ..., n] \), \( v > \in q_e(I) \).

5.2.2 OCL-Object and OCL-Value Translation

Algorithm 2 defines the translation of an OCL-Object or OCL-Value expression into a relational query that, intuitively, returns the object/value referred by the expression. Similarly as before, each tuple \( (t_1, ..., t_m, v) \) returned by the query represents an evaluation that the OCL_{FO} variables appearing in the expression might take (values \( t_1, ..., t_m \), taken from a context query \( q_e \)), together with the value retrieved by the OCL-Object/OCL-Value expression according to that evaluation \( v \).

Again, the idea behind the translation is to use the relational algebra operator that corresponds to those defined in the OCL_{FO} semantics. E.g., role navigations are translated by means of the same join we have defined in the OCL_{FO} semantics.

For instance, consider the OCL-Single expression \( m \cdot \text{sentTime} \) with a context query \( q_e = \tau_{neg}(\text{Group} \cdot \text{IsSentTo}) \) defining the values for the OCL_{FO} variable \( m \). Such expression is translated as:

$$\tau_{neg}(\text{Group} \cdot \text{IsSentTo}) \cdot \text{sentTime}$$

Intuitively, the translation converts the attribute navigation as a new join to retrieve the \text{sentTime} attribute for each value that \( m \) might take.

In the following, we formally proof the correctness of Algorithm 2.

Algorithm 2 raTranslation(OCL-Single, q_e)

if OCL-Single = \text{Constant} then
    return q_e \times \{\text{Constant}\}
else if OCL-Single = \text{Variable} then
    return q_e
else if OCL-Single = \text{Object}, \text{attr} then
    q_1 := raTranslation(OCL-Object_1, q_e)
    return π(q_1 \ \\ M \ attr)
else if OCL-Single = \text{Object}, \text{assocClass} then
    q_1 := raTranslation(OCL-Object_1, q_e)
    return π(q_1 \ \\ M \ assocClass)
end if

Theorem 3 Let \( \phi \) be an OCL-Single over a UML class diagram \( D_{\text{UML}} \), \( \overline{q} \) a relational algebra expression defined
over the relational view of $\text{SUM}_{L}$, and $q_{c}$ a context query such that $q = \text{raTranslation}(\phi, q_{c})$ (Algorithm 2). Then, for any interpretation $I$, and any substitution $S$, obtained from $q_{c}(I)$, we have:

$$\text{NULL} = \phi_{[S]}_{1} \iff \exists v. <t[0], \ldots, t[n], v> \in \overline{q}(I)$$

and, for any value $v$ different from NULL:

$$v = \phi_{[S]}^{I} \iff <t[0], \ldots, t[n], v> \in \overline{q}(I)$$

Proof The proof is based on structural induction over the OCL$_{F0}$ grammar. In the following we bring the proof for one base case and one inductive case. The rest of cases follows analogously. Consider the base case:

$$\phi = \text{self}$$

According to Algorithm 2,

$$\overline{q} = \text{raTranslation}(\phi, q_{c}) = q_{c} \quad (1)$$

According to the semantics, we have that $v = \text{self}_{[S]}^{I}$ iff $\text{self}_{[S]} = v$. This is the case if and only if we have $<t[0], \ldots, t[n], v> \in q_{c}$. Consider the inductive case:

$$\phi = \text{OCL-Object}.f\text{Attr}$$

According to Algorithm 2,

$$\overline{q} = \pi(q_{1} \land f\text{Attr})$$

where

$$q_{1} = \text{raTranslation}(\text{OCL-Object}, q_{c})$$

According to the semantics, we have that $v = (\text{OCL-Object}.f\text{Attr})_{[S]}^{I}$ iff $v = \pi (\text{OCL-Object}_{[S]} \land f\text{Attr}^{I})$. This is the case if and only if there exists some value $v'$ in OCL-Object$^{I}$ whose join with $f\text{Attr}^{I}$ retrieves $v$. By induction hypothesis we know that $<t[0], \ldots, t[n], v'> \in q(I)$ iff $v' = \text{OCL-Object}_{[S]}^{I}$. Thus, $v = (\text{OCL-Object}.f\text{Attr})_{[S]}^{I}$ iff $<t[0], \ldots, t[n], v> \in \overline{q}(I)$. □

5.2.3 OCL-Bool Translation Algorithm

In Algorithm 3 we show how to make use of the previous translations to obtain the values that cause the violation of the OCL-Bool condition. The output of this algorithm is a query that returns the evaluation of the OCL$_{F0}$ variables alive in OCL-Bool that make the expression evaluate to FALSE.

The intuition behind the translation is to use the relational algebra selection $\sigma$ to select those values that contradict the OCL-Bool expression.

For instance, consider the OCL-Bool expression

$$\text{self.user->forAll(u|not a = u)}$$

with the context query $q_{c} = \pi_{\text{author}}.\text{author}(\text{Group} \land \text{IsSentTo} \land \text{Writtes})$ defining the values for the OCL$_{F0}$ variable a depending on the value given to self. Such expression would be translated as:

$$\sigma_{\text{author}=\text{user}}(q_{c} \land (q_{c} \land \text{HasMember}))$$

Intuitively, $q_{c}$ retrieves the values that might take for every value of self (that is, all the authors of messages sent to some group self), then, $(q_{c} \land \text{IsMemberOf})$ retrieves the values that u might take for every value of self (that is, all the users of some group self). Then, the join of both expressions retrieves the values that a and u might take for the same value of self (that is, all the authors and members of some group self). Finally, the selection picks those tuples in which the value for a is equal to the value for u.

Note that, to translate the other normalized OCL$_{F0}$ boolean operations such as and and not we just need to compose the previous translation pattern. That is, and is translated by unifying the set of rows that causes the violation of the first condition, with those causing the violation of the second one; and not is translated by computing those rows violating the inner expression (in other words, those rows satisfying its negation), and removing them from the context query (so we have those rows violating the negated statement).

In the following, we formally proof the correctness of Algorithm 3.

### Algorithm 3 raTranslation(OCL-Bool, $q_{c}$)

```plaintext
if OCL_Boo = OCL_Bool1 and OCL_Boo2 then
    q1 = raTranslation(OCL_Boo1, $q_{c}$)
    q2 = raTranslation(OCL_Boo2, $q_{c}$)
    return $q_{1} \lor q_{2}$
else if OCL_Boo = not OCL_Boo1 then
    return $\neg q_{1}$ = raTranslation(OCL_Boo1, $q_{c}$)
else if OCL_Boo = OCL_Set->forAll(var|OCL_Boo1) then
    $q_{c}$ := raTranslation(OCL_Set, $q_{c}$)
    return $\neg q_{c}$
else if OCL_Boo1 = OCL_Single1 and OCL_Single2 then
    $q_{1}$ = raTranslation(OCL_Single1, $q_{c}$)
    $q_{2}$ = raTranslation(OCL_Single2, $q_{c}$)
    return $\neg q_{1} \land q_{2}$
else if OCL_Boo1 = OCL_Value then
    $q_{1}$ = raTranslation(OCL_Value1, $q_{c}$)
    return $\neg q_{1}$
end if
```

### Theorem 4 Let $\phi$ be an OCL-Bool over a UML class diagram $\text{SUM}_{L}$, $q$ a relational algebra expression defined over the relational view of $\text{SUM}_{L}$, and $q_{c}$ a context query such that $q = \text{raTranslation}(\phi, q_{c})$ (Algorithm 3). Then,
for any interpretation \( I \), and any substitution \( S_t \) obtained from \( q_c(I) \), we have:
\[
\phi[S_t] = \text{FALSE} \iff t \in \overline{q}(I)
\]

Proof: The proof is based on structural induction over the OCL\( \_\text{FO} \) grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. For the base case, we consider the expression:
\[
\phi = \text{OCL\_Single}_1 \text{ CompOp } \text{OCL\_Single}_2
\]

According to Algorithm 3,
\[
\overline{q} = \pi \sigma(q_1 \land q_2)
\]
where
\[
q_1 = \text{raTranslation(}\text{OCL\_Single}_1, q_c)\]
\[
q_2 = \text{raTranslation(}\text{OCL\_Single}_2, q_c)
\]

According to the semantics, \( \phi[S_t] = \text{FALSE} \iff \) the values \( v_1 = \text{OCL\_Single}_1[S_t] \), and \( v_2 = \text{OCL\_Single}_2[S_t] \) do not satisfy the comparison operator \( \text{CompOp} \). By induction hypothesis we have that \( t[0], \ldots, t[n], v_1 \in q_1(I) \), and \( t[0], \ldots, t[n], v_2 \in q_2(I) \). Thus, we can obtain the values \( v_1 \) and \( v_2 \) by joining \( q_1 \) and \( q_2 \), and select the row \( t[0], \ldots, t[n] \) iff \( v_1 \) and \( v_2 \) do not satisfy the corresponding \( \text{CompOp} \). Thus, \( \phi[S_t] = \text{FALSE} \iff t \in \overline{q}(I) \).

For the inductive case, we consider:
\[
\phi = \text{OCL\_Set}\_\to\text{forall}(\text{Var}\_\text{OCL\_Bool})
\]

According to Algorithm 3,
\[
\overline{q} = \pi q_b
\]
where
\[
q_b = \text{raTranslation(}\text{OCL\_Bool}, q_c)\]
\[
q_b = \text{raTranslation(}\text{OCL\_Set}, q_c)
\]

According to the semantics, we have that
\[
\text{OCL\_Set}\_\to\text{forall(Var}\_\text{OCL\_Bool})[S_t] = \text{FALSE} \iff \text{not OCL\_Bool}[S_t] = \emptyset \text{, this is the case iff } \not\exists v. v \in \text{OCL\_Set}[S_t] \text{ and } \text{OCL\_Bool}[S_t, (\text{Var}/v)] = \text{FALSE} \text{. By induction hypothesis, we have that, for any value } v, v \in \text{OCL\_Set}[S_t] \text{ if and only if } t[0], \ldots, t[n], v \in q_b(I) \text{. Moreover, by induction again we know that } v \in \text{OCL\_Set}[S_t] \text{ and } \text{OCL\_Bool}[S_t, (\text{Var}/v)] = \text{FALSE if and only if } t[0], \ldots, t[n], v \in q_b(I) \text{. Thus, } \text{OCL\_Set}\_\to\text{forall(Var}\_\text{OCL\_Bool})[S_t] = \text{FALSE if and only if } t \in \overline{q}(I) \text{.} \]

5.2.4 Translating an OCL\( \_\text{FO} \) Constraint

To translate an OCL\( \_\text{FO} \) constraint, it is enough to invoke Algorithm 3 with the body of the constraint as the OCL\( \_\text{Bool} \) parameter and the context class in which the constraint is defined as the context query \( q_c \), so that, the variable \text{self} \ is going to be evaluated to all the objects of the given context class.

For instance, consider the constraint of our running example MessagesAreFromGroup. This constraint would be translated into:
\[
\pi(\text{Group}\Join\text{IsSentTo}\Join\text{Votes}) \land \\
\pi\sigma((\text{Group}\Join\text{IsSentTo}\Join\text{Votes}) \Join \\
(\text{Group}\Join\text{IsSentTo}\Join\text{Votes}\Join\text{HasMember}))
\]

Intuitively, the query picks up all users who have written in some group (first line of the translation), and it takes out all those users who are indeed members of such group (second and third line of the translation). Thus, note that the constraint is satisfied if and only if the previous query is empty.

In the following we formally proof the correctness of the translation.

Theorem 5 Let \( \phi \) be an OCL\( \_\text{FO} \) constraint over a UML class diagram \( S_{\text{UML}} \), defined on the context class \( R \), and \( \overline{q} \) a relational algebra expression, defined over the relational view of \( S_{\text{UML}} \), such that \( \overline{q} = \text{raTranslation}(\phi, R) \) (Algorithm 3). Then, for any interpretation \( I \), we have:
\[
I \models \phi \iff \overline{q}(I) = \emptyset
\]

Proof: Directly from Theorem 4 we have that \( \overline{q}(I) \) retrieves those values for the OCL\( \_\text{FO} \) variable \text{self} from \( R^t \) s.t. that makes the OCL\_Bool in \( \phi \) evaluate to \text{FALSE}. Thus, \( I \models \phi \iff \overline{q}(I) = \emptyset \).

It is worth noting that the translation algorithm is linear w.r.t. the number of OCL operators and OCL variables of the constraint. Indeed, every OCL operator is translated for a fixed number of RA operations (1 in most cases, 3 in the worst case), and every OCL variable reference is translated as a RA query whose size is limited by the size of the OCL source it ranges from. Thus, the final size of the translation of an OCL constraint defined by o operations and v variables can be upper-bounded by \( 3o + 3ov \). Since every OCL operator is traversed only once in the algorithm, this result is computed in the same upper-bound amount of time. In any case, this computation is absolutely independent from the data, which means that data-complexity, the time that takes to analyze an OCL\( \_\text{FO} \) constraint w.r.t. the amount of data stored in the information system, is not affected by it. Intuitively, this is because this
translation can be seen as a compilation step to apply only once at the beginning, and reuse forever.

6 RA queries to OCL\textsubscript{FO} Constraints

Now, we show that any constraint that can be checked by means of a relational algebra query can be encoded as an OCL\textsubscript{FO} constraint. That is, for any relational query $q$, we can build an OCL\textsubscript{FO} constraint $\phi$ such that, for any interpretation $I$, we have that $q(I) = \emptyset$ iff $I \models \phi$.

This result implies that the language of OCL\textsubscript{FO} is as expressive for defining constraints as relational algebra. Taking into account that in Theorem 5 we showed that any OCL\textsubscript{FO} constraint can be checked by means of a relational query, we may conclude that OCL\textsubscript{FO} is exactly as expressive for defining constraints as relational algebra. Thus, checking OCL\textsubscript{FO} constraints is as difficult as executing a relational query, that is polynomial with regard to data complexity (and AC$^0$ in particular).

We define the $\text{oclTranslation}$ from a RA query to an OCL\textsubscript{FO} constraint in Algorithm 4. This algorithm receives three input parameters: a context query $q_c$, an OCL\textsubscript{FO} Boolean statement $\text{OCL-Bool}$, and a mapping $M$ that makes explicit which attributes from $q_c$ are mapped to which OCL\textsubscript{FO} variables from $\text{OCL-Bool}$. Then, the idea is that the algorithm returns a new OCL\textsubscript{FO} Boolean statement $\phi$ such that, for any valid interpretation $I$, $I \models \phi$ iff $\text{OCL-Bool}^M[I] = \text{true}$ for any valid substitution $S_i$ obtained from $q_c(I)$.

Then, we can then obtain the OCL\textsubscript{FO} constraint corresponding to a relational query $q$ by invoking $\text{oclTranslation}(q, \text{false}, \emptyset)$. Indeed, the unique way that for all substitutions $S_i$ obtained from $q_c$ we have $\text{false}^M[S_i] = \text{true}$ is that $q(I) = \emptyset$.

Intuitively, the algorithm works by recursively removing relational operators from the input query $q$ and placing them in the OCL-Boolean expression given as a parameter. For instance, if we invoke:

$$\text{oclTranslation}(R \setminus S, \text{false}, \emptyset)$$

We first recursively translate:

$$\text{oclTranslation}(S, x \leftrightarrow s, \{s \rightarrow S\})$$

to obtain an OCL\textsubscript{FO} Boolean expression that characterizes those values of some variable $x$ that are different to any element in $S$. In particular, we obtain the new OCL-Boolean $\phi$:

$$\text{S.allInstances}() \rightarrow \text{forall}(s | x \leftrightarrow s)$$

Then, we recursively translate:

$$\text{oclTranslation}(R, \phi \text{ implies false}, \{r \rightarrow R\})$$

obtaining:

$$\text{R.allInstances}() \rightarrow \text{forall}(r | \text{S.allInstances}() \rightarrow \text{forall}(s | x \leftrightarrow s) \text{ implies false})$$

This boolean expression iterates through all elements $x$ of $R$, it checks whether $x$ is different from every $s$ in $S$, and if so, it returns false. Clearly, such OCL-Boolean statement only evaluates to true iff the relational query $R \setminus S$ evaluates to the empty set.

Algorithm 4 makes use of some auxiliary functions. Function $\text{GetComponentMap}(M, q)$ returns a copy of the map $M$ but adding some new correspondences between relational attributes in $q$ and new fresh OCL\textsubscript{FO} variables. For instance, this function allowed us to create the new OCL free variables $r$ and $s$, and map them to the relational tables $R$ and $S$ respectively. Function $\text{getInequalities}(M_1, M_2, q_1, q_2)$ returns a conjunction of OCL\textsubscript{FO} variable inequalities. In particular, one inequality for each pair of OCL\textsubscript{FO} variables that are mapped to the same i-th relational attribute in $q_1$ and $q_2$, respectively. This function allowed us to build the inequality $r \leftrightarrow s$ in our previous example.

Theorem 6 Let OCL\textsubscript{Bool} be an OCL\textsubscript{FO} boolean statement defined over a UML class diagram $S_{UML}$, $q_c$ a relational algebra expression defined over the relational view of $S_{UML}$, and $M$ a mapping from $q_c$ to attributes to OCL\textsubscript{FO} variables in $\phi_1$. Let $\phi$ be the OCL\textsubscript{FO} statement such that $\phi = \text{oclTranslation}(q_c, \text{OCL-Bool}, M)$ (Algorithm 4). Then, for any interpretation $I$, and any substitution $S$, we have:

$$I \models \phi[S] \text{ iff } \forall_{S, \text{OCL-Bool}^M[S]}[S] = \text{true}$$

where the substitutions $S_i$ are obtained from the context query $q_c$ and the mapping $M$. 

Proof The proof is inductive on the number of relational algebra operators present in the input context query q, thus, following the recursive nature of the Algorithm.

For the base case, consider a context query with the following form, where \( R \) is the name of some relation:

\[ q_c = R \]

Applying Algorithm 4 we get:

\[ \phi = R.\text{allInstances}() \rightarrow \text{forall}(r \mid \text{OCL-Bool}) \]

Then, according to the semantics, for any substitution \( S_i \), we have that \( I \models R.\text{allInstances}() \rightarrow \text{forall}(r \mid \text{OCL-Bool}) \mid S \) if and only if \( \text{OCL-Bool}^f_{[r/v][S]} = \text{FALSE} \). This is the case iff for all the values \( v \in R^f \) we have \( \text{OCL-Bool}^f_{[r/v][S]} = \text{TRUE} \). Equivalently, this is the case iff for all the possible substitutions \( S_i \) we can obtain from \( R \), we have that \( \text{OCL-Bool}^f_{[S_i][S]} = \text{TRUE} \).

We deal now with the inductive cases. Consider first:

\[ q_c = q_1 \cup q_2 \]

Applying Algorithm 4 we get:

\[ \phi = \text{OCL-Bool}_1 \text{ and } \text{OCL-Bool}_2 \]

where

\[ \text{OCL-Bool}_1 = \text{oclTranslation}(q_1, \text{OCL-Bool}, \mathcal{M}) \]

According to the semantics, \( I \models \phi \mid S \) if and only if \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \) and \( \text{OCL-Bool}_2 \mid S = \text{TRUE} \). By induction we know that \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \) iff for every substitution \( S_i \) that can be obtained from \( q_1 \), we have \( \text{OCL-Bool}^f_{[S_i][S]} = \text{TRUE} \). Thus, \( I \models \phi \mid S \) if and only if \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \) for every substitution that can be obtained from \( q_1 \cup q_2 \).

Consider the case:

\[ q_c = \pi q_1 \]

Applying Algorithm 4 we get:

\[ \phi = \text{OCL-Bool}_1 = \text{oclTranslation}(q_1, \text{OCL-Bool}, \mathcal{M}) \]

And we have that \( I \models \phi \mid S \) iff \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \). By induction we know that \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \) iff for every substitution \( S_i \) that can be obtained from \( q_1 \), we have \( \text{OCL-Bool}^f_{[S_i][S]} = \text{TRUE} \). Thus, \( I \models \phi \mid S \) if and only if \( \text{OCL-Bool}_1 \mid S = \text{TRUE} \) for every substitution that can be obtained from \( \pi q_1 \).

Consider the case:

\[ q_c = \sigma_{a \neq b} q_1 \]

Applying Algorithm 4 we get:

\[ \phi = \text{oclTranslation}(q_1, \text{OCL-Bool}^{'}, \mathcal{M}^{'}) \]

where

\[ \text{OCL-Bool}^{'}, \text{ocl-ineq} + \text{‘implies’} + \text{OCL-Bool}, \mathcal{M}_2 \]

ocl-ineq = getInequalities(\( M_1, M_2, q_1, q_2 \))

\( M_1 = \text{getCompleteMap}(M, q_1) \)

\( M_2 = \text{getCompleteMap}(M, q_2) \)
By induction we know that \( I \models \phi[S] \) iff for any substitution \( S_1 \) obtained from \( q_1 \) it holds that \( \text{OCL-Bool} \downarrow_{S_1[S]} = \text{TRUE} \). Unfolding \( \text{OCL-Bool} \) using the induction hypothesis we have that \( I \models \phi[S] \) iff for any substitution \( S_2 \) from \( q_2 \), it holds that \( (\text{ocl-ineq}_{S_2[S]} \downarrow_{S_1[S]} = \text{TRUE} \). Equivalently, \( I \models \phi[S] \) iff for any substitution \( S_1 \) from \( q_1 \), and any substitution \( S_2 \) from \( q_2 \), it holds that \( \text{ocl-ineq}_{S_2[S]} \downarrow_{S_1[S]} = \text{FALSE} \) or \( \text{OCL-Bool} \downarrow_{S_1[S]} = \text{TRUE} \). Since we know that, for any substitution \( S_1 \) obtained from \( q_1 \) there is no substitution \( S_2 \) obtained from \( q_2 \) for which \( \text{ocl-ineq}_{S_2[S]} \downarrow_{S_1[S]} = \text{FALSE} \), we see that, \( I \models \phi[S] \) iff for any substitution \( S_1 \) obtained from \( q_1 \) \( \forall q_2 \) we have \( \text{OCL-Bool} \downarrow_{S_1[S]} = \text{TRUE} \). \( \Box \)

With this algorithm at hand, we now prove that any constraint that can be checked by means of relational algebra can be written in \( \text{OCL}_{FO} \).

**Theorem 7** Let \( S_{UML} \) be a UML class diagram, and \( q \) a relational algebra expression defined over the relational view of \( S_{UML} \). Then, consider the \( \text{OCL}_{FO} \) constraint \( \phi \) s.t. \( \phi = \text{oclTranslation}(q, \text{false}, \emptyset) \) (Algorithm 4). Then, for any interpretation \( I \), we have:

\[
I \models \phi \quad \text{iff} \quad q(I) = \emptyset
\]

**Proof** From Theorem 6 we know that \( I \models \phi \) iff for every substitution \( S_1 \) obtained from \( q \), \( \text{false}_S \downarrow_{S_1[S]} = \text{TRUE} \). This is the case if and only if there is no substitution \( S_1 \) obtained from \( q \). That is \( I \models \phi \) iff \( q(I) = \emptyset \). \( \Box \)

It is worth noting that the translation from RA to \( \text{OCL}_{FO} \) is quadratic with the number of RA operations. Indeed, most RA operators are translated into a fixed number of OCL operations (1 in most cases, 2 in the worst one), but the set-difference operator gives rise to \( k \) OCL equality operations, where \( k \) is the given number of columns of the relation where the difference is applied. Thus, the length of the translation of an RA query into OCL can be bounded by \( 2o + sK \), where \( o \) is the number of RA operators, \( s \) the number of set-difference operators, and \( K \) the maximum number of columns for the biggest table. It is easy to see also that the algorithm obtains this result in quadratic time since it visits every RA operator only once, and each RA operator is translated in constant time (if no subroutine is applied), or linear time (otherwise).

### 7 OCL\(_{CORE}\)

To conclude our analysis, we identify a minimal subset of \( \text{OCL}_{FO} \) that we call \( \text{OCL}_{CORE} \). \( \text{OCL}_{CORE} \) is minimal in the sense that any of its proper subsets is not sufficient to encode the whole \( \text{OCL}_{FO} \). Several minimal core fragments might exist.

The main practical relevance of \( \text{OCL}_{CORE} \) is that of allowing to compare the expressive power of other fragments of OCL that can be proposed in the future with that of \( \text{OCL}_{FO} \). Note that it is guaranteed that if a fragment contains the six operations in \( \text{OCL}_{CORE} \), then its expressive power will be at least that of \( \text{OCL}_{FO} \). For instance, using the \( \text{OCL}_{CORE} \) it is easy to realize that the work presented in [24] implements the whole expressiveness of \( \text{OCL}_{FO} \) since it implements the six basic operations. This is much easier to be determined than having to check whether the fragment contains all the operations in \( \text{OCL}_{FO} \).

Moreover, \( \text{OCL}_{CORE} \) could be used to define a UML base model [25]. That is, a minimal subset of UML/OCL features that do not loose expressiveness. In any case, we would like to stress that \( \text{OCL}_{CORE} \) is not intended to be used by the designers when specifying their models, since this would result in very complex and difficult to understand expressions for defining the integrity constraints, but for OCL researchers and developers.

In Figure 7 we define \( \text{OCL}_{CORE} \), which only contains the operations: allInstances, forAll, implies, \&, and =, together with the operation to navigate from a variable to a role/attribute.

\[
\begin{align*}
\text{OCL-Bool} &::= \text{OCL-Bool} \text{ implies } \text{OCL-Bool} | \\
\text{OCL-Set} &::= \text{OCL-Set} \text{ forAll } (\text{Var} | \text{OCL-Bool}) | \\
\text{OCL-Object} &::= \text{OCL-Object} = \text{OCL-Object} | \\
\text{OCL-Value} &::= \text{Var} | \text{role} | \\
\text{Var-attr} &::= \text{Var-attr} | (\text{a constant name})
\end{align*}
\]

**Fig. 7** Syntax of \( \text{OCL}_{CORE} \)

Next theorem proves that this fragment is able to encode all constraints in \( \text{OCL}_{FO} \).

**Theorem 8** For any \( \text{OCL}_{FO} \) constraint \( \phi \), there is an \( \text{OCL}_{CORE} \) constraint \( \phi_c \), such that, for any interpretation \( I \):

\[
I \models \phi \iff I \models \phi_c
\]

**Proof** Given any \( \phi \) constraint written in \( \text{OCL}_{FO} \), we can obtain its corresponding constraint \( \phi_c \) written in \( \text{OCL}_{CORE} \) by first translating \( \phi \) to a RA query \( q \), and then, translating \( q \) to \( \text{OCL}_{CORE} \). Any \( \text{OCL}_{FO} \) constraint can be translated into a relational algebra query \( q \) following the process described in Section 5. Then, we
can translate \( q \) back into OCL using the process described in Section 6, thus, obtaining an OCL\(_{FO} \) constraint \( \varphi' \). By construction, this \( \varphi' \) already accommodates the OCL\(_{CORE} \) syntax described in 7, except for operations and not that might appear in \( \varphi' \) but are not included in OCL\(_{CORE} \). However, we can easily get rid of these operations using common boolean equivalence with implies (e.g. not OCL\(_{-} Boo\)l is equivalent to OCL\(_{-} Boo\)l implies 1=2).

Now, it only lacks to show that OCL\(_{CORE} \) is minimal since we cannot remove any operation from it without loosing expressiveness. We cannot remove allInstances since it is the only operation that allows obtaining a set of instances from a UML class. We cannot take out forAll because it is the only operation that can be applied after allInstances. The implies operation is mandatory to encode, for instance, the OCL not. Finally, without \( \leq \), \( = \), or \( \neq \), we would not be able to encode \( \leq \) (among others).

It is worth noting that OCL\(_{CORE} \) is a proper subset of normalized OCL\(_{FO} \) which, in turn, is a proper subset of OCL\(_{FO} \). However, from these three fragments, only OCL\(_{FO} \) is intended for the use of the designers when specifying UML class diagrams with OCL constraints. It is worth mentioning also that the three variants have the same expressive power since both the normalized expression and the OCL\(_{CORE} \) expression of a constraint can be automatically drawn from its OCL\(_{FO} \) expression.

8 Related Work

We first discuss related work that studied the relationship between OCL and RA. Then we compare our OCL fragment with those fragments identified for the problem of OCL satisfiability. Finally, we compare our work with other suggested OCL translations.

8.1 OCL and Relational Algebra Studies

The expressiveness of OCL and its relationship with Relational Algebra has been previously discussed by Mandel and Cengarle in [6]. However, whereas Mandel and Cengarle addressed the expressive power of OCL as a query language, we look at OCL as a constraint language. The main difference is that a constraint language deals only with boolean expressions, so when looking at the equivalence of OCL w.r.t. RA, we focus on whether we can check some OCL constraint through checking the emptiness of a RA query \( q \), and vice versa. In contrast, Mandel and Cengarle investigate whether for every RA query \( q \) we can build an OCL expression that returns the same tuples as \( q \). In particular, the authors argue that this is impossible since (1) OCL has no tuple constructor, and (2) OCL has no way to dynamically create new types. However, in our equivalence notion of OCL with RA, tuples do not play a role. Indeed, we only need to check whether a true/false OCL expression can be mapped to an empty/non-empty RA query, whereas they tried to map any RA query into an equivalent OCL expression. In addition, Mandel and Cengarle hinted (although not formally proved) that OCL version 1 was not Turing-Complete. This claim has been proved to be no longer valid for OCL version 2.4 in our article.

Since OCL 2.0 introduced tuple facilities, Balsters argued that OCL is able to encode any RA query composed of the basic RA operations [26], that is: union, difference, product, renaming, selection and projection operations. However, Balsters stressed in his work that, still, OCL is not equivalent to RA in a maximal sense since it is impossible in OCL to define a new operation that receives as input two arbitrary sets of tuples, and outputs the natural join of them. Note that this supposed impediment does not affect us since our goal is to show that any given RA query can be rewritten into an equivalent OCL\(_{FO} \) (not necessarily in a generic way).

From a more practical point of view, Queralt and Teniente proposed in [27] a translation from OCL to domain-independent first-order logics, which is equivalent to relational algebra. The fragment covered in their translation is expressively equivalent to OCL\(_{FO} \) since they cover OCL\(_{CORE} \). Interestingly, their translation is also based on first normalizing an OCL constraint into another one composed of less expressions. Probably, a further study of such normalization might bring another OCL\(_{CORE} \) for OCL\(_{FO} \). However, since their intention was to apply first-order reasoners on OCL rather than discussing OCL expressiveness, they did not prove that domain-independent first-order logic statements were expressible in OCL, neither that such normalization could bring a core, as we have done.

Another translation of a fragment of OCL into first-order logic is proposed by Clavel et al. in [28]. By naturally extending their translation of inequalities to inequalities with objects, we can see that their OCL fragment covered is expressively equivalent to OCL\(_{FO} \). In contrast, the translation given by Beckert et al. in [29] seems to deal with a broader subset of OCL. However, their translation is not pure first-order logics since, for instance, it uses some built-in functions to count the number of times an object appears in a collection unrestrictedly, which is not a first-order capability.

It is important to note that none of these proposals departs from an OCL formal semantics. Thus, none
of the previous translations has a proof of soundness. It can be argued that no soundness proof is required since, in the absence of formal OCL semantics, the semantics of OCL turns to be the translation itself. However, using such translations as the semantics for OCL is cumbersome and error prone. Indeed, they are defined by means of multiple algorithms and functions. Thus, it is extraordinarily difficult to assess, for instance, whether the semantics given by Queralta and Temiente [27] is equivalent to the one given by Clavel et al. [28]. In contrast, the OCLF0 semantics we provide in this paper is concise and based on basic set theory (i.e., set inclusion, exclusion, etc). Then, it could be used to prove that some translation is sound w.r.t. OCLF0 semantics, and thus, two OCL translations would be equivalent if they are both sound with respect to OCLF0 semantics.

Finally, there are some tools that implement translations from OCL into SQL. Egea et al. introduced MySQLACL [30], which generates MySQL code for a subset of OCL expressions. However, the translation defined clearly falls out of RA since it uses MySQL specific procedures. Another tool, part of the well-known Dresden OCL Toolkit [10], is OCL2SQL. It produces a translation in standard SQL, but lacks theoretical basis for sophisticated cases. Indeed, the translation is based on some straightforward patterns without any formal proof [31], thus, it is not clear the correctness of the translation when dealing with, for instance, NULL values. Indeed, OCL2SQL makes use of SQL NOT EXISTS expressions which is known to have spurious behavior when dealing with NULL values, but no discussion on this aspect is given.

8.2 OCL subsets for satisfiability

OCLF0 is a fragment of OCL defined for ensuring efficient integrity checking. However, other fragments of OCL have been defined pursuing different objectives. One of the most prominent is finding OCL subsets for which the problem of OCL constraint satisfiability is decidable.

The problem of constraint satisfiability is the problem of ensuring whether there exists at least one instance of the UML class diagram such that satisfies the constraint. Since it is known that checking OCL constraints satisfiability is undecidable in general [8], several decidable OCL subsets have been identified. However, since the problem of constraint satisfiability is harder than the problem of constraint checking, those OCL subsets tend to be less expressive.

In our particular case, we have that OCLF0 is expressively equivalent to first-order logics. Thus, since it is known that checking the satisfiability of a first-order constraint is not decidable, OCL based languages for reasoning satisfiability cannot deal with full first-order logic expressiveness. Two examples of such languages are: OCL-Lite [17], and OCLUNIV [32].

OCL-Lite is a proper subset of OCLF0 expressively equivalent to the ALCT Description Logics. Essentially, it is the part of OCLF0 that excludes any kind of OCL operator that could be used to emulate maximum cardinalities (e.g. \( \rightarrow size() < k \)). Indeed, to ensure that OCL-Lite constraint satisfiability is decidable, OCL-Lite must be applied to UML class diagrams without maximum cardinalities.

OCLUNIV is also a proper subset of OCLF0. In this case, it is based on the language of weakly acyclic TGDs [33]. The rational behind OCLUNIV is that it excludes any kind of OCL operator that could be used to emulate minimum cardinalities (e.g. \( \rightarrow notEmpty() \), \( \rightarrow size() > k \)). This is because OCLUNIV decidability is only ensured over UML class diagrams with no minimum cardinality cycles.

Thus, it seems that, to ensure decidability for the satisfiability problem, one has to give up either maximum cardinalities, or minimum cardinalities. However, both can be easily taken into account for the problem of checking constraints (indeed, OCLF0 can encode both).

Figure 8 summarizes the relationship between OCLF0, OCL-Lite, and OCLUNIV.

![Fig. 8 Relation of different OCL fragments](image)

8.3 Other OCL translations

There are some tools in the literature that translates OCL into other languages. These tools are mainly aimed to the problem of OCL constraint satisfiability, and ensure its decidability by means of a bounding the search space (e.g. limiting the size of the UML instance to search for).

For instance, the approaches presented in [34–36] proposes a translation from OCL into the Alloy/Kodkod languages so that, satisfiability can be checked through the Alloy/Kodkod reasoner. It is worth to mention that these tools can deal with the OCL \( \rightarrow closure \) operation, which is outside first-order logics, and thus, outside OCLF0. Furthermore, more ex-
pressive translations based on Constraint-Satisfaction Problem [37], and Satisfiability Modulo Theories [38] have also been proposed. An easy way to see that these translations are more expressive is that, on one hand, they implement all the operations from OCLCore, and they provide some operations not inside OCLFO (such as the comparison of the size of two sets, which is not a first-order expression).

However, these approaches are only focused on reasoning constraint satisfiability, and are not thought for bringing formal semantics into OCL, neither showing efficient satisfaction checking, as we do.

Indeed, we bring formal semantics based on set-theory. These semantics describe the evaluation of OCL expressions in a brief and unambiguous manner as can be seen in Figure 5. Thus, these semantics can be used to check-proof the correctness of OCL translations (as we do with the translation between OCL and RA). However, the previous translations are not checked/proved on any particular OCL semantics. Thus, it is impossible to know how does such translations interpret OCL expressions (neither if they are pairwise equivalent) without studying the semantics of the particular language they use, which are not even standard in the case of Alloy/Kodkod.

9 Conclusions

OCL is a formal language for defining constraints that serves as a complement for graphical modelling languages such as UML. However, full OCL is so expressive that checking OCL constraints is not even semidecidable, as we have shown in this paper.

To tackle this issue, we have identified OCLFO, the fragment of OCL equivalent to RA. That is, any OCLFO constraint can be evaluated by checking if some RA query is empty (which guarantees efficiency), and any RA query can be translated into OCLFO (which guarantees expressiveness).

The syntax and semantics of OCLFO are defined in a concise way and thus, we argue that they can be easily adopted by OCL practitioners. Moreover, we identify the minimal subset of OCLFO with its same expressive power, OCLCore, which makes OCLFO an easy object of study.

As future work, we plan to exploit the equivalence between OCLFO and RA to adapt the techniques from the database field to the OCL and conceptual modeling community. In the first place, we would like to study an automatic efficient translation from OCLFO to SQL including optimizations such as indexes, or better encodings of the associations into relational tables. This translation could also benefit from incremental query techniques (thus, obtaining incremental integral checking), and view updating proposals (thus, obtaining OCLFO incremental maintenance). Indeed, some proposals treating OCL problems through database techniques have recently been published with exciting results [23,14,16]. Furthermore, we would like to study if we can include OCL Bags, OrderedSets, and Sequences into OCLFO, for instance, by means of adapting the techniques described in [39].

Acknowledgements This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2017-87610-R, and by the Secretaría d’Universitat i Recerca de la Generalitat de Catalunya under 2017 SGR 1749.

References


